OBJECT ORIENTED RUBY

Ruby is pure object-oriented language and everything appears to Ruby, as an object. Every value in Ruby is an object, even the most primitive things: strings, numbers and even true and false. Even a class itself is an *object* that is an instance of the *Class* class. This chapter will take you through all the major functionalities related to Object Oriented Ruby.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and methods within a class are called members of the class.

Ruby class definition:

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the **class name** and is delimited with an **end**. For example we defined the Box class using the keyword class as follows:

```
class Box
   code
end
```

The name must begin with a capital letter and by convention names that contain more than one word are run together with each word capitalized and no separating characters (CamelCase).

Define ruby objects:

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class using **new** keyword. Following statements declare two objects of class Box:

```
box1 = Box.new
box2 = Box.new
```

The initialize method:

The **initialize method** is a standard Ruby class method and works almost same way as **constructor** works in other object oriented programming languages. The initialize method is useful when you want to initialize some class variables at the time of object creation. This method may take a list of parameters and like any other ruby method it would be preceded by **def** keyword as shown below:

```
class Box
   def initialize(w,h)
      @width, @height = w, h
   end
end
```

The instance variables:

The **instance variables** are kind of class attributes and they become properties of objects once we objects are created using the class. Every object's attributes are assigned individually and share no value with other objects. They are accessed using the @ operator within the class but to access them outside of the class we use **public** methods which are called **accessor methods**. If we take above defined class **Box** then @width and @height are instance variables for the

class Box.

```
class Box
   def initialize(w,h)
      # assign instance avriables
      @width, @height = w, h
   end
end
```

The accessor & setter methods:

To make the variables available from outside the class, they must be defined within **accessor methods**, these accessor methods are also known as a getter methods. Following example shows the usage of accessor methods:

```
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end

   # accessor methods
   def printWidth
      @width
   end

   def printHeight
      @height
   end
end

# create an object
box = Box.new(10, 20)

# use accessor methods
x = box.printWidth()
y = box.printHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

When the above code is executed, it produces following result:

```
Width of the box is : 10
Height of the box is : 20
```

Similar to accessor methods which are used to access the value of the variables, Ruby provides a way to set the values of those variables from outside of the class using **setter methods**, which are defined as below:

```
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end

   # accessor methods
   def getWidth
      @width
   end
   def getHeight
      @height
```

```
   end

   # setter methods
   def setWidth=(value)
      @width = value
   end
   def setHeight=(value)
      @height = value
   end
end

# create an object
box = Box.new(10, 20)

# use setter methods
box.setWidth = 30
box.setHeight = 50

# use accessor methods
x = box.getWidth()
y = box.getHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

When the above code is executed, it produces following result:

```
Width of the box is : 30
Height of the box is : 50
```

The instance methods:

The **instance methods** are also defined in the same way as we define any other method using **def** keyword and they can be used using a class instance only as shown below. Their functionality is not limited to access the instance variables, but also they can do a lot more as per your requirement.

```
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # instance method
   def getArea
      @width * @height
   end
end

# create an object
box = Box.new(10, 20)

# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"
```

When the above code is executed, it produces following result:

```
Area of the box is : 200
```

The class methods & variables:

The **class variables** is a variable which is shared between all instances of a class. In other words, there is one instance of the variable and it is accessed by object instances. Class variables are prefixed with two @ characters (@@). A class

variable must be initialized within the class definition as shown below.

A class method is defined using **def self.methodname()** which ends with end delimiter and would be called using class name as **classname.methodname** as shown in the following example:

```
class Box
   # Initialize our class variables
   @@count = 0
   def initialize(w,h)
      # assign instance avriables
      @width, @height = w, h

      @@count += 1
   end

   def self.printCount()
      puts "Box count is : #@@count"
   end
end

# create two object
box1 = Box.new(10, 20)
box2 = Box.new(30, 100)

# call class method to print box count
Box.printCount()
```

When the above code is executed, it produces following result:

```
Box count is : 2
```

The to_s Method:

Any class you define should have a **to_s** instance method to return a string representation of the object. Following a simple example to represent a Box object in terms of width and height:

```
#!/usr/bin/ruby -w

class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # define to_s method
   def to_s
      "(w:#@width,h:#@height)"  # string formatting of the object.
   end
end

# create an object
box = Box.new(10, 20)

# to_s method will be called in reference of string automatically.
puts "String representation of box is : #{box}"
```

When the above code is executed, it produces following result:

```
String representation of box is : (w:10,h:20)
```

Access Control:

Ruby gives you three levels of protection at instance methods level which may be **public, private, or protected**. Ruby does not apply any access control over instance and class variables.

- **Public Methods:** Public methods can be called by anyone. Methods are public by default except for initialize, which is always private.
- **Private Methods:** Private methods cannot be accessed, or even viewed from outside the class. Only the class methods can access private members.
- **Protected Methods:** A protected method can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.

Following is the simple example to show the syntax of all the three access modifiers:

```
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end

   # instance method by default it is public
   def getArea
      getWidth() * getHeight
   end

   # define private accessor methods
   def getWidth
      @width
   end
   def getHeight
      @height
   end
   # make them private
   private :getWidth, :getHeight

   # instance method to print area
   def printArea
      @area = getWidth() * getHeight
      puts "Big box area is : #@area"
   end
   # make it protected
   protected :printArea
end

# create an object
box = Box.new(10, 20)

# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"

# try to call protected or methods
box.printArea()
```

When the above code is executed, it produces following result. Here first method is called successfully but second method gave a problem.

```
Area of the box is : 200
test.rb:42: protected method `printArea' called for #
<Box:0xb7f11280 @height=20, @width=10> (NoMethodError)
```

Class Inheritance:

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

Inheritance also provides an opportunity to reuse the code functionality and fast implementation time but unfortunately Ruby does not support Multiple level of inheritances but Ruby supports **mixins**. A mixin is like a specialized implementation of multiple inheritance in which only the interface portion is inherited.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base class or superclass**, and the new class is referred to as the **derived class or sub-class**.

Ruby also supports the concept of subclassing ie. inheritance and following example explains the concept. The syntax for extending a class is simple. Just add a **<** character and the name of the superclass to your class statement. For example, following define a class *BigBox* as a subclass of *Box*:

```
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # instance method
   def getArea
      @width * @height
   end
end

# define a subclass
class BigBox < Box

   # add a new instance method
   def printArea
      @area = @width * @height
      puts "Big box area is : #@area"
   end
end

# create an object
box = BigBox.new(10, 20)

# print the area
box.printArea()
```

When the above code is executed, it produces following result:

```
Big box area is : 200
```

Methods Overriding:

Though you can add new functionality in a derived class, but sometime you would like to change the behavior of already defined method in a parent class. You can do so simply by keeping the method name same and overriding the functionality of the method as shown below in the example:

```
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # instance method
   def getArea
      @width * @height
   end
end
```

```
# define a subclass
class BigBox < Box

   # change existing getArea method as follows
   def getArea
      @area = @width * @height
      puts "Big box area is : #@area"
   end
end

# create an object
box = BigBox.new(10, 20)

# print the area using overriden method.
box.getArea()
```

Operator Overloading:

We'd like the + operator to perform vector addition of two Box objects using +, the * operator to multiply a Box width and height by a scalar, and the unary - operator to do negate the width and height of the Box. Here is a version of the Box class with mathematical operators defined:

```
class Box
  def initialize(w,h) # Initialize the width and height
    @width,@height = w, h
  end

  def +(other)          # Define + to do vector addition
    Box.new(@width + other.width, @height + other.height)
  end

  def -@                # Define unary minus to negate width and height
    Box.new(-@width, -@height)
  end

  def *(scalar)         # To perform scalar multiplication
    Box.new(@width*scalar, @height*scalar)
  end
end
```

Freezing Objects:

Sometimes we want to prevent an object from being changed. The freeze method in Object allows us to do this, effectively turning an object into a constant. Any object can be frozen by invoking **Object.freeze**. A frozen object may not be modified: you can't change its instance variables.

You can check if a given object is already frozen or not using **Object.frozen?** method which returns true in case object is frozen otherwise a false value is return. Following example clears the concept:

```
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end

   # accessor methods
   def getWidth
      @width
   end
   def getHeight
      @height
   end
```

```
   # setter methods
   def setWidth=(value)
      @width = value
   end
   def setHeight=(value)
      @height = value
   end
end

# create an object
box = Box.new(10, 20)

# let us freez this object
box.freeze
if( box.frozen? )
   puts "Box object is frozen object"
else
   puts "Box object is normal object"
end

# now try using setter methods
box.setWidth = 30
box.setHeight = 50

# use accessor methods
x = box.getWidth()
y = box.getHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

When the above code is executed, it produces following result:

```
Box object is frozen object
test.rb:20:in `setWidth=': can't modify frozen object (TypeError)
        from test.rb:39
```

Class Constants:

You can define a constant inside a class by assigning a direct numeric or string value to a variable which is defined without using either @ or @@. By convention we keep constant names in upper case.

Once a constant is defined, you can not change its value but you can access a constant directly inside a class much like a variable but if you want to access a constant outside of the class then you would have to use **classname::constant** as shown in the below example.

```
#!/usr/bin/ruby -w

# define a class
class Box
   BOX_COMPANY = "TATA Inc"
   BOXWEIGHT = 10
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # instance method
   def getArea
      @width * @height
   end
end

# create an object
box = Box.new(10, 20)

# call instance methods
a = box.getArea()
```

```
puts "Area of the box is : #{a}"
puts Box::BOX_COMPANY
puts "Box weight is: #{Box::BOXWEIGHT}"
```

When the above code is executed, it produces following result:

```
Area of the box is : 200
TATA Inc
Box weight is: 10
```

Class constants are inherited and can be overridden like instance methods.

Create object using allocate:

There may be a situation when you want to create an object without calling its constructor **initialize** ie. using new method, in such case you can call allocate which will create an uninitialized object for you as in the following example:

```
#!/usr/bin/ruby -w

# define a class
class Box
   attr_accessor :width, :height

   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end

   # instance method
   def getArea
      @width * @height
   end
end

# create an object using new
box1 = Box.new(10, 20)

# create another object using allocate
box2 = Box.allocate

# call instance method using box1
a = box1.getArea()
puts "Area of the box is : #{a}"

# call instance method using box2
a = box2.getArea()
puts "Area of the box is : #{a}"
```

When the above code is executed, it produces following result:

```
Area of the box is : 200
test.rb:14: warning: instance variable @width not initialized
test.rb:14: warning: instance variable @height not initialized
test.rb:14:in `getArea': undefined method `*'
   for nil:NilClass (NoMethodError) from test.rb:29
```

Class Information:

If class definitions are executable code, this implies that they execute in the context of some object: self must reference something. Let's find out what it is.

```
#!/usr/bin/ruby -w

class Box
   # print class information
```

```
   puts "Type of self = #{self.type}"
   puts "Name of self = #{self.name}"
end
```

When the above code is executed, it produces following result:

```
Type of self = Class
Name of self = Box
```

This means that a class definition is executed with that class as the current object. This means that methods in the metaclass and its superclasses will be available during the execution of the method definition.